In Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'10), pp. 27-31, March 2010, published as Technical Report No.33, Hasso-Plattner Institut, University of Potsdam

AspectKE*: Security Aspects with Program Analysis for Distributed Systems

Fan Yang
DTU Informatics,Technical University of Denmark
fy@imm.dtu.dk

Hidehiko Masuhara
Graduate School of Arts and Sciences,University of Tokyo
masuhara@acm.org

Tomoyuki Aotani
Graduate School of Arts and Sciences,University of Tokyo
aotani@graco.c.u-tokyo.ac.jp

Flemming Nielson
DTU Informatics,Technical University of Denmark
nielson@imm.dtu.dk

Hanne Riis Nielson
DTU Informatics,Technical University of Denmark
riis@imm.dtu.dk

Abstract

Enforcing security policies to distributed systems is difficult, in particular, when a system contains untrusted components. We designed AspectKE*, a distributed AOP language based on a tuple space, to tackle this issue. In AspectKE*, aspects can enforce access control policies that depend on future behavior of running processes. One of the key language features is the predicates and functions that extract results of static program analysis, which are useful for defining security aspects that have to know about future behavior of a program. AspectKE* also provides a novel variable binding mechanism for pointcuts, so that pointcuts can uniformly specify join points based on both static and dynamic information about the program. Our implementation strategy performs fundamental static analysis at load-time, so as to retain runtime overheads minimal. We implemented a compiler for AspectKE*, and demonstrate usefulness of AspectKE* through a security aspect for a distributed chat system.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; D.4.6 [*Security and Protection*]: Access controls; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Design, Languages, Security

Keywords Aspect Oriented Programming, Program Analysis, Security Policies, Distributed Systems, Tuple Spaces

1. Introduction

Enforcing security policies to a distributed system is challenging, especially when trusted components of a system have to work with untrusted components. In such a case, we need to ensure that untrusted components do not break security policies of the system. A common approach is to statically check security properties of the untrusted components before their execution [7, 9]. For example, Java type checks downloaded code before execution.

The approach has two problems. The first is lack of flexibility: the programmers cannot easily (re)define security policies, as they are normally integrated with a compiler and runtime system of the language. The second is expressiveness: static analyses are sometimes too restrictive to accurately enforce security policies in practice, as they have to approximate properties of a program, and cannot be combined with runtime information.

In order to address those problems, we designed and implemented AspectKE*, an aspect-oriented programming (AOP) language based on a tuple space system. AspectKE* has the following key characteristics.

- It provides high-level program analysis predicates and functions that can be used as pointcuts in aspects. Since those predicates and functions give information on future behavior of processes, the programmers can easily apply aspects (e.g., security aspects) to processes that are defined by untrusted parties.
- It also provides a novel variable binding mechanism for pointcuts, so that the programmers can specify static and dynamic conditions in a uniform manner.
- Its implementation strategy realizes runtime evaluation of program analysis predicates and functions with minimal runtime overheads, which is achieved by analyzing the required static information beforehand at the load-time, and merely looking it up at runtime.
- It is the first AOP system that is based on a tuple space. Even though tuple space based systems are not predominant in the industry, we believe that our techniques can be applied to other distributed systems as well.

The rest of this paper is organized as follows. Section 2 describes the problems that we address. Section 3 outlines our design principles for solving the problem. Section 4 proposes our AOP language. Section 5 shows our solution to the problems in Section 2. Section 6 sketches implementation issues. Sections 7 discusses the related work and Section 8 concludes the paper.

2. Motivating Problem

Imagine a company that runs a chat system for exchanging messages among its employees. In order for the employees to access the system from outside the company, the chat system allows client programs (a third-party software) to be executed on untrusted computers. Now the challenge is how to ensure secrecy and integrity of data exchanged between company and employees, especially when we cannot trust client processes running on a computer with lower-security.

First, let us see a chat system without any security mechanism. Figure 1 illustrates a simplified distributed chat system that consists of six nodes. The nodes ALICE and BOB represent two users' chat functionalities inside the chat system (trusted). The nodes GUI1 and GUI2 represent the users who use the chat system (trusted). The nodes CLIENT1 and CLIENT2 represent third-party software

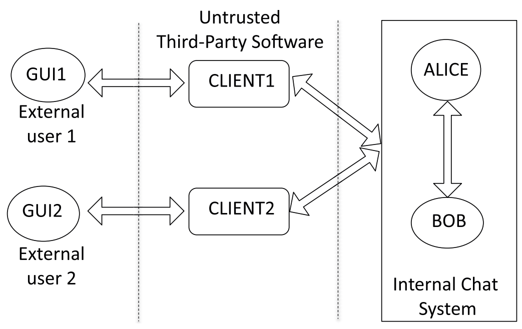

Figure 1. An Overview of a Simplified Chat System

running on untrusted computers that relay messages between users and chat function nodes (untrusted).

Let us focus on CLIENT1 and CLIENT2, as they are the only untrusted parts. Besides performing intended operations, a third-party client might contain malicious code that performs unintended operations. Listings 1 and 2 show a code fragment of node CLIENT1, written in AspectKE*, implementing a user login procedure. Line 8 of Listing 2 is added to the original client definition so that it will leak password information to an eavesdropper.

```
1  node CLIENT1{
2      process clientlogin(CLIENT1,GUI1);
3  }
```

Listing 1. Node CLIENT1

```
1  proc clientlogin(location self,location gui){
2      location user;
3      symbol password;
4
5      in(OUTPUTG,LOGIN,user,password)@gui;
6      out(INPUTU,LOGIN,password,self)@user;
7
8      out(LOGIN,user,password)@EAVESDROPPER;
9
10     in(OUTPUTU,LOGIN,user)@self;
11     out(INPUTG,LOGIN,user,SUCCESS)@gui;
12
13     parallel{
14         process clientsendmsg(self,user,gui);
15         process clientreceivemsg(self,user,gui);
16     }
17 }
```

Listing 2. Process clientlogin

Listing 1 defines the node CLIENT1, which instantiates a process clientlogin, with CLIENT1 and GUI1 as parameters. Constants are capitalized in this paper, whose declarations are omitted in this paper.

Listing 2 defines process clientlogin. Lines 2-3 define local variables. Line 5 waits for an input of a user name and a password information from a user. For example, when Alice (using GUI1) inputs a login request, Line 5 binds the variable user to the user name (i.e., ALICE), and the variable password to the password typed in (e.g., ALICEPW). Line 6 sends the password information to the corresponding user node at the server computer by referencing the two variables. A process (omitted here) at the chat function node will send a confirmation message if the password is correct. Lines 10-11 receive the confirmation message and notify GUI1 of successful login. Lines 13-15 start processes for handling messages between users (details of this step are not discussed in the paper).

The definition of CLIENT1 except for Line 8 is intended; i.e., it performs no malicious operations. The operation at Line 8 is additional malicious code that sends user and password to node EAVESDROPPER.

To ensure the secrecy and integrity of users' data which pass through untrusted components, we pose the following security policy: CLIENT1 is allowed to get data from GUI1 only when the obtained data is sent to the specified trusted nodes. For the program above, the policy means that the input action at Line 5 is permitted only if its continuation process does not send password to any node other than user. This security policy essentially demands to perform static analysis of the continuation process (Lines 6-17) before actually performing the input action (Line 5). In this paper, we show how to integrate the static analysis techniques into a security aspect with minimal runtime overheads.

3. Design Principles

3.1 Static Analysis for Security Aspect

Some security policies need information on future events. An example is the security policy mentioned in Section 2, where we cannot decide whether to permit an input action performed by an untrusted process without inspecting how the password information will be used in the future. In this paper, we integrate static analysis techniques into aspect definition, and provide an expressive way of specifying security aspects that refer to future events.

3.2 Program Analysis Predicates and Functions

The language for composing security aspects should have comprehensive interface for using static analysis techniques. We provide several high-level program analysis predicates and functions that extract static analysis results of a program, so that the users can easily specify security policies in aspects. In addition, our novel variable binding mechanism for pointcuts enables the programmers to specify static and dynamic conditions in a uniform manner.

3.3 Load-Time Static Analysis

We perform static analysis at load-time because it fits a distributed setting and can retain runtime overheads minimal as well. In principle, static analysis can be performed at either compile time, load-time, or run-time. However, compile-time analysis requires the definition of processes which is not realistic in a distributed system with mobile processes. Run-time analysis is not feasible either as it causes huge runtime overheads.

4. AspectKE*

We designed and implemented AspectKE* programming language, an aspect extension to the Klava tuple space system[2]. Since Klava is a distributed tuple space system (DTS), we briefly introduce basic concept in DTS.

A DTS consists of *nodes*, *processes*, *tuple spaces* and *tuples*. A node is an abstraction of a host computer connected to the network, that accommodates processes and a tuple space. A tuple space is a repository of tuples that can be accessed concurrently from processes. A process is a thread of execution that can output (through out action) its data as a tuple to a tuple space, and can retrieve (through read or in action) data from a tuple space by matching a pattern. Unlike classical tuple space systems such as Linda [5] that assume a globally shared tuple space, a DTS contains shared tuple spaces distributed over a network. Besides the standard actions about retrieving and outputting tuples on a local or remote nodes, a Klava process can also create new processes on a local or remote node (through eval action), can create a new remote node (through newloc action).

In AspectKE*, aspects are global activities that monitor actions performed by all processes in a Klava system.

4.1 The Hello World Example

Listing 3 shows a Hello World program that demonstrates the basic usage of nodes and processes. In the program, a process at node N1 reads HELLO and WORLD from its own tuple space and create a process at node N2 that outputs these words in a different order.

```
 1  location N1,N2;
 2  symbol W1,W2,HELLO,WORLD;
 3
 4  node N1{
 5      data (N2,W1,HELLO);
 6      data (N2,W2,WORLD);
 7      process p1(N2);
 8  }
 9
10  node N2{
11  }
12
13  proc p1(location baz){
14      symbol foo, bar;
15      read(baz,W1,foo)@N1;
16      in(baz,W2,bar)@N1;
17      eval(process p2(foo,bar,baz))@baz
18  }
19
20  proc p2(symbol foo,symbol bar,location baz){
21      out(foo,bar)@N2;
22      out(bar,foo)@baz;
23  }
```

Listing 3. Hello World Program

Lines 1 and 2 declare constants. The type *location* is a set of logical node locations. The type *symbol* is a set of globally distinguishable data. Lines 4-11 define initial states of node N1 and N2. Node N1 consists of two tuples and one process. Node N2 is empty. Lines 13-18 define a process p1. Line 14 declares two local variables, which are bound to values by an input action. For example, the tuple ⟨baz,W1,foo⟩ at Line 15 matches the tuple ⟨N2,W1,HELLO⟩ at node N1, and binds foo to HELLO. Line 16 performs an in action, which reads a tuple ⟨N2,W2,WORLD⟩ from N1 in a similar manner to read actions, and then removes the read tuple. Line 17 creates a process p2 with parameters HELLO, WORLD and N2 at node N2. The process p2 then executes two out actions that output HELLO and WORLD onto the node N2 in a different orders.

4.2 A Simple Aspect for Hello World

Listing 4 defines a simple aspect that prevents read actions in the Hello World program from executing. Note that in AspectKE*, all actions are joint points.

```
 1  aspect a1(N1)
 2  {  read(location VAR n, symbol VAR w,
 3          symbol FORMAL word)@(N1)
 4      -> process z;
 5      { case(!w=W1) break;
 6        case(!beused(word,z)) break;
 7        case(forall(x,targeted(OUT,z))<x=n>) break;
 8        default proceed;
 9      }
10  }
```

Listing 4. A Simple Aspect

4.2.1 Pointcut

Lines 1-4 define a pointcut that captures a read action (which reads N1's tuple space) performed at node N1. Parameters of the pointcut specify types (either location or symbol) and kinds (either VAR or FORMAL). When the joint point (Line 15 in Listing 3) is to be executed, variables n and w are bound to values N2 and W1, respectively. The variable word, whose kind is *formal*, is bound to a variable foo in the process.

Note that a *formal* variable is bound to a variable in a process, unlike the binding mechanism of *var* variable and formal parameters of an advice declaration in AspectJ, which are bound to values. This idea is originally proposed in our previous work [6] in order to deal with open joinpoints that extensively occur in tuple space systems. We adopt this mechanism for specifying usage of variables in a process that is not yet bound to any value when an action is performed.

The description at Line 4 binds the variable z to a continuation process right after the captured action. When the pointcut matches the read action at Line 15 in Listing 3, z denotes actions performed by Lines 16-18 and 20-23.

In addition, our variable binding mechanism can internally link static information to each variable, thus enables programmers to specify static and dynamic conditions regarding the bound variables in a uniform manner.

4.2.2 Advice

Lines 5-9 define a piece of advice that terminates an executing process if one of the following three conditions holds. (1) Its second parameter is not equal to W1. (2) Its third parameter will not be used in the rest of the process. (3) All out actions in the rest of the process target at the location specified by the first parameter n. Each case statement consists of a condition and suggestion (*break* or *proceed*). If *break* is executed, the current process stops. If *proceed* is executed, the current process continues. When pointcut matches the join point as mentioned in Section 4.2.1, the action is terminated by the third case.

From the advice definition, it is obvious that the first case condition does not hold. The second case condition uses a *program analysis predicate* beused, which does not hold as well. In AspectKE*, *program analysis predicates (and functions)* are language constructs for aspects that predict future behavior of a program. Here we only explain them by examples, but their definition will be discussed in the next subsection. This beused predicate checks future behavior of an executing process, namely, whether variable foo (captured by word) is not referenced in any action of the continuation processes. Since it uses foo in the out actions at Line 21 and 22 of Listing 3, the beused predicate holds, which in turn makes the overall condition false. Note that the aspect has to check the condition before executing those out actions. This means that we need to analyze the future behavior of a program.

The third case is complicated, although the expression itself looks quite simple thanks to our novel binding mechanism. It checks whether the first argument is used as the destination of all out actions in the continuation process by a predicate forall and a *program analysis function* targeted.

All destination locations of out actions in the process z are collected and returned as a set by the function targeted(OUT,z). For example, if z contains two out actions out(...)@c and out(...)@v, targeted(OUT,z) returns a set {c,v}. Each element in the set is either a constant (e.g., N2 at Line 21 in Listing 3) or a variable name (e.g., baz at Line 22 in Listing 3).

Predicate & Function	*The Return Value*
performed(z)	returns the set of all actions that are performed in z
targeted(OUT,z)	returns the set of all destination locations of out actions in z
beused(foo,z)	returns true if variable foo is used in any actions of z
beused(foo,OUT,z)	returns true if variable foo is used in out actions of z
beusedsafe(foo,OUT, A,z)	returns true if variable foo either will not be used in out actions of z at all, or used in out actions of z, but only be performed to locations within set A.

Table 1. Program Analysis Predicates and Functions

The predicate forall(x,A)<x=n> holds when all the elements in A is equal to n. Note that equality is checked in a different ways depending on what x denotes. If x denotes a concrete value, x=n is true when n equals to the value. If x denotes a variable v, x=n is true when v will be bound to n if proceeded. When matching the joint point at Line 15 in Listing 3, A is a set that contains the constant N2 and the variable baz whose runtime value is also N2, which in turn lets the pointcut binds n to N2 as well. Thus forall(x,A)<x=n> is true according to the definition of the forall predicate and equality. The aspect will then suggests *break* in order to terminate the read action. Since both runtime data and static information are needed to evaluate this condition, it goes beyond a static property of n. It also shows that the programmers can specify static and dynamic conditions of n in a uniform manner.

4.3 Program Analysis Predicates and Functions

Table 1 summarizes the program analysis predicates and functions in AspectKE*, where foo is a bound formal variable; OUT is a type of actions (can be replaced with other types of actions); z is a the continuation process of the captured action; and A is a collection of locations which includes *locations* in two forms: constants and bound formal (or var) variables. These predicates and functions are designed to specify different properties of the continuation program.

5. A Security Aspect for the Distributed Chat System

Listing 5 presents a security aspect with program analysis predicates to enforce the security policy presented in Section 2: the input action at Line 5 in Listing 2 is permitted only if its continuation process will never output password to any node other than user.

```
1  aspect in_loginpw(location VAR s){
2     in(OUTPUTG,LOGIN, location FORMAL uid ,
3       symbol FORMAL pw)@(location VAR gui)
4       -> process z;
5     { case(element_of(gui,{GUI1,GUI2})&&
6             !beusedsafe(pw,OUT,{uid},z))
7             break;
8       default
9             proceed;
10    }
11 }
```

Listing 5. Aspect for Protecting Password Usage

Upon the joint point at Line 5 in Listing 2, the var variables s and gui are bound to CLIENT1 and GUI1, respectively; the formal variables uid and pw are bound to the variable user and password, respectively. At Line 5 in Listing 5, the predicate element_of returns true since GUI1 is in the set {GUI1,GUI2}. At Line 6, the program analysis predicate beusedsafe checks if the continuation process z outputs password only to location user. Since the underlying static analysis detects that password is output to EAVESDROPPER, this predicate returns false. Thus the overall suggestion from the advice is to *break*, which results in termination of the malicious client.

6. Implementation Issues

Our AOP system consists of a translator from AspectKE* to Java and a runtime system that supports tuple space and AOP operations. The translator translates a source program in AspectKE* into a Java program that exploits distributed operations in a runtime library. The runtime system matches and executes aspects dynamically so that new security policies can be applied to a running system.

In order to efficiently evaluate a program analysis predicate and function in an aspect, our system performs context insensitive interprocedural dataflow analyses when it dynamically loads process definitions. The analyzer takes process definitions at the Java bytecode level, in order to apply aspects to a system without source code, which is the common approach in the distributed mobile processes.

Figure 2 shows how the program analysis predicates and functions work with advice. The runtime system matches each action with pointcut descriptions. When matches, it evaluates a program analysis predicate (or function) by looking up the result performed at load-time. We developed a mapping mechanism that associates bound variables in aspects to static information of the bound value from analysis results. For example, when evaluating the beused(word,z) predicate at Line 6 of Listing 4, it picks up the relevant static analysis results (variable foo at Line 15 links with variables foo appeared in Line 21 and 22 of Listing 3) to evaluate whether word (mapping to foo at Line 15 of Listing 3) is used in z (the continuation process of Line 15 in Listing 3).

Regarding the performance, since the load-time static analysis does not incur much runtime cost, we believe that our approach is practical and can be used in other AOP systems that need to check the future behavior of programs.

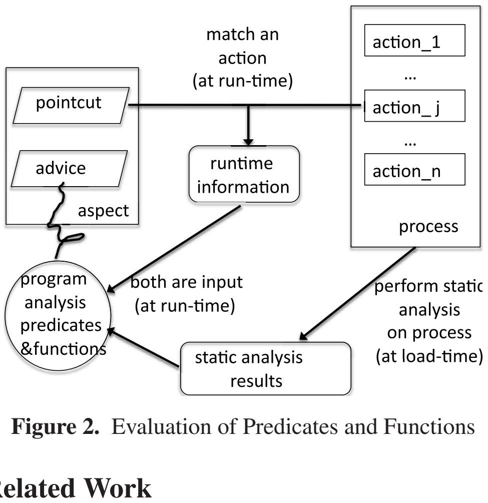
match an action (at run-time)
pointcut
advice
aspect
runtime information
action_1
...
action_ j
...
action_n
process
program analysis predicates &functions
both are input (at run-time)
perform static analysis on process (at load-time)
static analysis results

Figure 2. Evaluation of Predicates and Functions

7. Related Work

There are several tuple space systems that provide a certain security mechanism. For example, KLAIM[10] (with Klava[2] as its implementation) uses a static type system to realize access control. SECOS[12] provides a low-level security mechanism that protects every tuple field with a lock. JavaSpaces[4], which is used in industrial contexts, has a security mechanism based on the Java security framework. Our work is different from these in combining aspects with program analysis techniques, hence provides more flexible and precise ways to specify and enforce security policies.

There are several AOP systems in which pointcuts can specify relationships between join points. AspectJ's cflow pointcut captures join points based on a control flow in a program, which can be used for implementing access control mechanisms. Dataflow pointcut [8] identifies joint points based on flow of the information, which can be used for enforcement of secrecy and integrity. However, both pointcuts capture control or data flow that have happened before, rather than in the future. Some advanced AOP languages [1, 3, 11] allow the programmers to define their own pointcut primitives, including those that exploit program analysis results. In theory, it is also possible for those languages to define security aspects based on the future behavior of a program by defining pointcuts that statically analyze the program. However, those languages offer accesses to the programs at bytecode or AST-level, which makes it hard to implement correct and efficient static analyses.

Our approach, in contrast, provides predicates and functions that give relatively high-level information about future behavior, which makes it much easier to implement security aspects. Additionally, due to the novel binding mechanism of variables in pointcuts, our language is more expressive for specifying analysis properties.

8. Conclusions

We designed and implemented a prototype of AspectKE* that can retrofit existing, or even running distributed systems by applying security aspects. As an AOP system, our contributions can be summarized as follows. (1) AspectKE* can straightforwardly express a large set of security policies, especially those based on future behavior of executing processes. (2) The high-level program analysis predicates and functions allow the programmers to directly specify security policies without defining complicated program analysis. (3) The novel variable binding mechanism for pointcuts enables aspects to express dynamic properties of an executing process in combination with static properties derived by the static analysis predicates and functions. (4) We proposed an efficient implementation strategy that combines load-time static analysis and runtime checking, so as to keep runtime overheads minimal while keeping expressiveness of aspects.

Current AspectKE* can merely monitor processes and command the processes to *break* or *proceed* from its advice. We plan to extend the language so that it can perform other kind of actions. The challenge is how to formulate static analysis as aspects can introduce extra data- and control-flows into processes that should also be monitored by static analysis predicates and functions.

Acknowledgments

This work is partly supported by the Danish Strategic Research Council (project 2106-06-0028) "Aspects of Security for Citizens". We would like to thank Lorenzo Bettini for discussing about the Klava system, Christian Probst, Hubert Baumeister, and Sebastian Nanz for their early comments, and the members of the PPP group at the University of Tokyo, Robert Hirschfeld and his research group members for their comments on the work.

References

[1] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD'07*, pages 161–172. ACM, 2007. ISBN 1-59593-615-7.

[2] L. Bettini, R. D. Nicola, and R. Pugliese. Klava: a Java package for distributed and mobile applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[3] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. In *AOSD'04*, pages 102–111. ACM, 2004.

[4] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces principles, patterns, and practice*. Addison-Wesley Longman Ltd. Essex, UK, UK, 1999.

[5] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. ISSN 0164-0925.

[6] C. Hankin, F. Nielson, H. R. Nielson, and F. Yang. Advice for coordination. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 153–168. Springer, 2008.

[7] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification*. Addison-Wesley Professional, 1999.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS'03*, volume 2895 of *LNCS*, pages 105–121. Springer, 2003.

[9] G. C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119. ACM, 1997.

[10] R. D. Nicola, G. L. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[11] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05*, volume 3586 of *LNCS*, pages 214–240. Springer, 2005.

[12] J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Sci. Comput. Program.*, 46(1-2):163–193, 2003.